GENERATION OF OPTIMIZED TOP-K SQL QUERY

M.Boopathiraj[1], R.Hari Krishnan[2], R.Rajmohan[3]
Dept of Computer Sciene & Engg, Thiagarajar College of Engineering Madurai, Tamil Nadu, India

ABSTRACT

Database usage is growing rapidly in recent years due to the development in internet and cloud usage. Because of this answering to why some of the expected tuples are missing in query execution has received more attention. In this paper we deal with the answering of why not question in sql query with SPJA (ie, selection, projection, join, aggregation) constructs and providing a refined query that will include the missing tuples. This includes the tuple that contains both numerical and non-numerical attributes. Given the original SQL query and expected tuple, the algorithm will return the refined query whose result will include the expected tuple with the result and its penalty. The algorithm tries all possible combinations of changes on the original query like changing the SPJA construct ,K-value, weighting vector or combinations of the above and it compares the result with the expected one .If it matches ,then it calculate the penalty for the respective query and return the one with least value.

Keywords: why not question, missing tuple, penalty, refined query

I. INTRODUCTION

A why-not question is being posed when a user wants to know why her expected tuples do not show up in the query result. Currently, end users cannot directly sift through the dataset to determine "why-not?" because the query interface (e.g., web forms) restricts the types of query that they can express. When end users query the data through a database application and ask "why-not?" but do not find any means to get an explanation through the query interface, that would easily cause them to throw up their hands and walk away from the tool forever—the worst result that nobody, especially the database application developers who have spent months to build the database applications, want to see. Unfortunately, supporting the feature of explaining missing answers requires deep knowledge of various database query evaluation algorithms, which is beyond the capabilities of most database application developers. In view of this, recently, the database community has started to research techniques to answer why-not questions on various query types. Among them, a few works have focused on answering why-not questions on Select-Project-Join-Aggregate (SPJA) SQL queries (e.g., [1], [2], [3], [4]) and preference queries (e.g., top-k queries [5], reverse skyline queries). So far, these proposals only work independently. For example, when answering why-not questions on top-k queries, the proposal in [9] assumes there are no SPJ constructs (e.g., selection, projection, join, and aggregation).

In this paper, we study the problem of answering why not top-k questions in the context of SQL. Generally, a top-k query in SQL appears as:

```
SELECT <ATTRIBUTES> FROM <TABLE NAME> WHERE <COND> AND ORDER BY <WEIGHTING_VECTOR>AND < LIMIT (K) VALUE>
```

Where,
ATTRIBUTES = A1, A2, A3 …AM, AGGREGATE (.)
TABLE NAME = T1 …Tk
COND = JOIN PREDICATE OR SELECTION PREDICATE.WEIGHING_VECTOR = WEIGHT VALUES

To address the problem of answering why-not questions on top-k SQL queries, we employ the query refinement approach [4], [5],

Specifically, given as inputs the original top-k SQL query and a set of missing tuples, this approach requires to return to the user a refined query whose result includes the missing tuples as well as the original query results. In this paper, we show that finding the best refined query is actually computationally expensive.

Afterwards, we present efficient algorithms that can obtain the best approximate explanations (i.e., the refined query) in reasonable time. We present case studies to demonstrate our solutions. We also present experimental results to show that our solutions return high quality explanations efficiently. This paper is an extension of [5], which discussed answering why-not questions on top-k queries in the absence of other SQL constructs such as selection, projection, join, and aggregation.

Related work:

Explaining a null answer for a database query was set out by but the concept of why-not was first formally discussed in [1]. That work answers a user's why-not question on Select-Project-Join (SPJ) queries by telling her which query operator(s) eliminated her desired tuples. After that, this line of work has gradually expanded. In [2] and [3], the missing answers of SPJ [2] and SPJA [3] queries are explained by a data-refinement approach, i.e., it tells the user how the data should be modified (e.g., adding a tuple) if she wants the missing answer back to the result. In [4], a query-refinement approach is adopted. The answer to a why-not question is to tell the user how to revise her original SPJA queries so that the missing answers can return to the result. They define that a good refined query should be (a) similar — have few "edits" comparing with the original query (e.g., modifying the constant value in a selection predicate is a type of edit; adding/ removing a join predicate is another type of edit) and (b) precise — have few extra tuples in the result, except the original result plus the missing tuples. In this paper, we adopt the query-refinement approach as our explanation model and also apply the above similarity and precision metrics

II. BASIC CONCEPTS

As an example, consider table U in Fig. 1a and the following top-3 SQL query:

```
Qo:    SELECT U.ID FROM U
WHERE U.A 205
ORDER BY 0.5 * U.A + 0.5 * U.B
LIMIT 3
```

Fig. 1b shows the ranking scores of all tuples in U and the top-3 result is {P3, P1, P2}. Assuming that we are interested in asking why P5 is not in the top-3, we see that using SPJA query modification techniques in [8] to modify only the SPJ constructs (e.g., modifying WHERE clause to be U.A140) cannot include P5 in the top-3 result (because P5 indeed ranks 4th under the current weighting w~ ¼j0:5 0:5j). Using our preliminary top-k query modification technique [9] to modify only the top-k constructs (e.g., modifying k to be four) cannot work either because P5 is filtered by the WHERE clause. This motivates us to develop holistic solutions that consider the modification of both SPJA constructs and top-k constructs in order

a) Example table U:

ID	A	B
P1	240	60
P2	235	60
P3	340	70
P4	100	70
P5	**140**	**100**
P6	150	50

b) Ranking under original weightings

ID	0.5*A + 0.5*B
P3	205
P1	150
P2	147.5
P5	150
P6	100
P4	85

To answer why-not questions on top-k SQL queries. For the example above, the following refined query Q^1 is one candidate answer:

```
Q1:
SELECT U.ID FROM U
WHERE U.A 140
ORDER BY 0.5 * U.A + 0.5 * U.B
LIMIT  4
```

Q^0 is precise because it includes no extra tuple and is similar to Q_1 because only essential edits were carried out: (1) modifying from U.A205 to U.A140, and (2) Modifying k from three to four

III. DESCRIPTION
3 ANSWERING WHY-NOT QUESTIONS ON TOP-K SPJ QUERIES

In this section, we first focus on answering why-not questions on top-k SQL queries with SPJ clauses. We will extend the discussion to why-not top-k SQL queries with GROUP BY and AGGREGATION in the next section.

3.1 The Problem and The Explanation Model

We consider a top-k SPJ query Q with a set of Select-Project Join clauses SPJ, a monotonic scoring function f and a weighting vector w~ ¼jw½1 w½2 w½dj, where d is the number of attributes in the scoring function. For simplicity, we assume a larger value means a better score (and rank) and the weighting space subject to the constraint P w½i¼ 1 where 0 w½i 1. We only consider conjunctions of predicates $P_1 \wedge ... \wedge P_n$, where each P_i is either a selection predicate "A_j op v" or a join predicate "A_j op A_k", where A is an attribute, v is a constant, and op is a comparison operator. For simplicity, our discussion focuses on comparison because generalizing our discussion to other comparison operators is straightforward. The query result would then be a set of k tuples whose scores are the largest

(In case tuples with the same scores are tied at rank kth, only one of them is returned).

Initially, a user issues an original top-k SPJ query Q_oðSPJ$_o$; ko; w~$_o$Þ on a dataset D. After she gets the query result, denoted as R_o, she may pose a why-not question with a set of missing tuples Y ¼fy_1;...;y_lg (l 1), where y_i has the same set of projection attributes as Q_o. In this paper, we adopt the query-refinement approach in [8] so that the system returns the user a refined query Q^0ðSPJ^0;k^0;w~^{0}Þ, whose result R^0 includes Y and R_o, i.e., fY [R_og R^0. It is possible that there are indeed no refined queries Q^0 that can include Y (e.g., Y contains a missing tuple whose expected attribute values indeed do not exist in the database). For those cases, the system will report to the user about her error.

There are possibly multiple refined queries for being the answers to a why-not question hQ_o;Y i. We thus use DSPJ, Dk, and Dw to measure the quality of a refined query Q^0, where Dk ¼ k^0 k$_o$, Dw ¼jjw~^{0} w~$_o$jj2, and DSPJ is defined based on four different types of edit operations of SPJ clauses adopted in [8]:

Following [8], we do not allow other edit operations such as changing the projection attributes (because users usually have a clear intent about the projection attributes). Note that there is no explicit edit operation for removing a selection predicate, since it is equivalent to modifying the constant value in the predicate to cover the whole domain of the attribute. Furthermore, we also do not consider modifying the joins to include self-join. Let c_i denote the cost of the edit operation e_i, and we follow [8] to set c_1 ¼ 1;c_2 ¼ 3; c_3 ¼ 5;c_4 ¼ 7. So, DSPJ ¼ P1i4ðc_in$_i$Þ, where n_i is the number of edit operations e_i used to obtain the refined query Q^0. In order to capture a user's tolerance to the changes of SPJ clauses, k, and w~ on her original query Q_o, we first define a basic penalty model that sets the penalties $_{spj}$, $_k$ and $_w$ to DSPJ, Dk and Dw, respectively, where spj þ k þ w ¼ 1:

Note that the basic penalty model is able to capture both the similar and precise requirements. Specifically, a refined query Q^0 that minimizes Basic_Penalty implies it is similar to the original query Q_o. To make the result precise (i.e., having fewer extra tuples), we can set a larger penalty $_k$ to Dk such that modifying k significantly is undesired.

The basic penalty model, however, has a drawback because Dk generally could be a large integer (as large as jDj) whereas Dw and DSPJ are generally smaller. One possible way to mitigate this discrimination is to normalize them respectively.

[Normalizing DSPJ] We normalize DSPJ using the maximum editing cost $DSPJ_{max}$.

Definition 1 (maximum editing cost $DSPJ_{max}$).

Given the original query Q_o, the maximum editing cost $DSPJ_{max}$ is the editing cost of obtaining a refined SPJ query $Q^{SPJ}{}_{max}$, whose (1) SPJ constructs most deviated from the SPJ constructs of the original query Q_o (based on the four types of edit operations e_1 to e_4) and (2) with a query result that includes all missing tuples Y and the original query result R_o

Example 1 (maximum editing cost $DSPJ_{max}$).

```
SELECT B FROM T1, T2
WHERE T1.A = T2.A AND D 400
ORDER BY 0.5 * D + 0.5 * E
LIMIT 2
```

By referring to Fig. 3 (the join result of T_1 ffl T_2), the result R_o of the top-2 query is: {Gary, Alice}. Assuming the missing tuples set Y of the why-not question is

{Chandler}. Then, $Q^{SPJ}{}_{max}$ is:[1]

```
QSPJmax:
SELECT B FROM T1, T2, T3
WHERE T1.A = T2.A AND T1: A ¼ T3: A
AND C 50 AND D 100
AND E 50
AND F 60 AND G 200 AND H  60
```

Definition 2 (worst rank with minimal edits). The worst rank with minimal edits r_o is the worst rank among all tuples in Y [R_o of a refined top-k SQL query $Q^{SPJ}{}_{min}$, whose (1) SPJ constructs least deviated from the original query Q_o (measured by c_1 to c_4), (2) using the original weighting $w\sim_o$, (3) with a query result that includes all missing tuples Y and the original query result R_o, and (4) the modification of k is minimal.

To explain why r_o is a suitable value to normalize Dk, we first remark that we could normalize Dk using the cardinality of the join result of Q_o because that is the worst possible rank. But to get a more reasonable normalizing constant, we look at Equation (1). First, to obtain the "worst" but reasonable value of Dk, we can assume that we do not modify the weighting, leading to condition (2) in Definition 2. Similarly, we do not want to modify the SPJ constructs so much but we hope the SPJ constructs at least do not filter out the missing tuples Y and the original query result R_o, leading to conditions (1) and (3) in Definition 2. So, based on Example 1, $Q^{SPJ}{}_{min}$ is:

```
QSPJmin:
SELECT B FROM T 1, T2 WHERE T1.A =
T2.A AND D 200
ORDER BY 0.5 * D + 0.5 * E LIMIT 7
```

We note that the following is not $Q^{SPJ}{}_{min}$ although it also satisfies conditions (1) to (3) because its modification of k is from two to eight, which not minimal (condition 4) is comparing with the true $Q^{SPJ}{}_{min}$ above:

```
SELECT B FROM T1, T2
WHERE T1.A = T2.A AND D 100
ORDER BY 0.5 * D + 0.5 * E LIMIT 8
```

The problem definition is as follows. Given a why-not question $hQ_o;Y$ i, where Y is a set of missing tuples and Q_o is the user's initial query with result R_o, our goal is to find a refined top-k SQL query Q^0ðSPJ^0;k^0;w~^{0}Þ that includes Y [R_o in the result with the smallest Penalty. In this paper, we use Equation (2) as the penalty function. Nevertheless, our solution works for all kinds of monotonic (with respect to all DSPJ, Dk and Dw) penalty functions. For better usability, we do not explicitly ask users to specify the values for $_{spj}$, $_k$ and $_w$. Instead, we follow our early work [9] so that users are prompted to answer a simple multiple-choice question as illustrated

Assume the default option "Never mind" is chosen. Table 2 lists some examples of refined queries that could be the answer of the why-not question to Q_o in Example 1. According to the above discussion, we have $DSPJ_{max}$ ¼ refined queries, $Q^0{}_1$ dominates $Q^0{}_2$ because its Dk is smaller than that of $Q^0{}_2$ and the other dimensions are equal. The best refined query in the example is $Q^0{}_4$ (Penalty ¼ 0:11). At this point, readers may notice that the best refined query is in the skyline of the answer space of three dimensions: (1) $DSPJ_i$, (2) Dk_i, and (3) Dw_i. Later, we will show how to exploit properties like this to obtain better efficiency in our algorithm. Penalty fumction is given by,

3. 2 PROBLEM ANALYSIS

Answering a why-not question is essentially searching for the best refined SPJ clauses and weighting in (1) the space SSPJ of all possible modified SPJ clauses and in (2) the space

2. The number of choices and the pre-defined values for spj, k and w, of course, could be adjusted. For example, to make the result precise (i.e., having fewer extra tuples), we suggest the user to choose the option where $_k$ is a large value.

TABLE 1
Examples of Candidate Refined Queries

(Option Never Mind (NM): $_{spj}$ ¼ 1=3, $_{k}$ ¼ 1=3, $_{w}$ ¼ 1=3)

Refined Query	SPJi	ki	wi	penalty
Q1({'T1 T2,D>=200},7,|0.5 0.5|)	1	5	0	0.35
Q2({'T1 T2,D>=100},8,|0.5 0.5|)	1	6	0	0.41
Q3({'T1 T2,D>=200},7,|0.6 0.4|)	1	5	0.14	0.38
Q4({'T1 T2,D>=200^C>=90}3,7,|0.5 0.5|)	4	1	0	0.11
Q5({'T1 T2,D>=200},3,|0.2 0.8|)	1	1	0.42	0.20

Fig1.T1 |><| T2 ranked under w~o ¼ j0:5 0:5j.

Sw of all possible weightings, respectively. It is not necessary to search for k because once the best set of SPJ clauses and the best weighting w~ are found, the value of k can be accordingly set as the worst rank of tuples in Y [Ro. The search space SSPJ can be further divided into two: (1a) the space of the query schemas SQS and (1b) the space of all the selection conditions Ssel. A query schema QS represents the set of relations in the FROM clause and the set of join predicates in the WHERE clause. A selection condition represents the set of selection predicates in the WHERE clause.

Consider Example 1 again. The query schema QS^0 that can include back the missing tuple (Chandler) would lead to a join result like Fig. 3. Originally, we have to consider eight predicates when dealing with attribute D, which are D 500, D 400, D 300, D 290, D 280, D 250, D 210, and D 100. By using Lemma 2, we just need to consider D 200 because among fyg[Ro ¼ { Chandler, Gary, Alice}, their attribute values of D are 200, 400, and 500, with 200 as the minimum. Similar for attribute E, by using Lemma 2, we just need to consider E 80. The above discussion can be straightforwardly generalized to other comparisons including , <, and >.

3.3 Skipping Progressive Top-k SQL Operations

In PHASE-2 of our algorithm, the basic idea is to execute progressive top-k SQL queries for all selection conditions in

S_{sel}^{QS0} and all weightings in Sw. After the discussion in Section 3.3.2, we know them some progressive top-k SQL executions can stop early. We now illustrate three pruning opportunities where some of those executions could be skipped entirely.

The first pruning opportunity is based on the observation from [15] that under the same selection condition seli, similar weighting vectors may lead to top-k SQL results with more common tuples. Therefore, if an operation TOPKðseli;w~j; until-see-ffyg[RogÞÞ for w~j has already been executed, and if a weighting w~l is similar to w~j, then we can use the query result Rij of topkðseli;w~j;until-see-ffyg[RogÞ to deduce the smallest k value for seli and w~l. Let k^0 be the deduced k value for seli and w~l. If the deduced k^0 is larger than the threshold ranking r^T, then we can skip the entire TOPKðseli;w~l;stopping-conditionÞ operation.

ID	Name
P1	Alice
P2	Bob
P3	Chandler
P4	Daniel
P5	Eagle
P6	Fabio
P7	Gray
P8	Henry

Fig1.Table T1

ID	A	B	C	CITY
P1	90	400	80	bangalore
P2	60	290	60	dindigul
P3	90	200	100	Salem
P4	50	300	70	bangalore
P5	80	100	210	salem
P6	70	250	70	madurai
P7	50	280	50	bangalore
P8	100	500	100	madurai

Fig2.Table T2

We illustrate the above by reusing our running example. Assume that we have cached the result sets of executed progressive top-k SQL queries. Let R_{1o} be the result set of the first executed query topkðsel1;w~o;until-see-ffyg[RogÞ and

R_o ¼ft5;t6g. Assume that R_{1o} ¼½t1;t2;t3;t4;t5;t6;y. Then, when we are considering the next weighting vector, say, $w_{\sim 1}$, in S_w, we first follow Equation (3) to calculate the threshold ranking r^T. In Fig. 5, projecting $w_{\sim 1}$ onto slope $P_{Q0}1o$

we get r^T ¼ 4 þ 2 ¼ 6. Next we calculate the scores of all tuples in R_{1o} using $w_{\sim 1}$ as the weighting. More specifically, let us denote the tuple in fyg[R_o under weighting vector $w_{\sim 1}$ as t_{bad} if it has the worst rank among fyg[R_o. In the example, assume under $w_{\sim 1}$, the scores of t_1, t_2, t_3 and t_4 are still better than t_{bad}, then the k^0 value for sel_1 and $w_{\sim 1}$ is at least 4 þ 3 ¼ 7. Since k^0 is worse than r^T ¼ 6, we can skip the entire TOPKðsel1;w~1;stopping-conditionÞ operation.

The above caching technique is shown to be the most effective between similar weighting vectors [15]. Therefore, we design the algorithm in a way that the list of weightings S_w is sorted according to their corresponding Dw_i values (of course, $w_{\sim o}$ is in the head of the list since Dw_o ¼ 0). In addition, the technique is general so that the cached result for a specific selection condition sel_i can also be used to derive the smallest k^0 value for another selection condition sel_j. As long as sel_i and sel_j are similar, the chance that we can deduce k^0 from the cached result that leads to TOPK operation pruning is also higher. So, we design the algorithm in a way that sel_i is enumerated in increasing order of Dsel as well.

The second pruning opportunity is to exploit the best possible ranking of fyg[R_o (under all possible weightings) to set up an early termination condition for some weightings, so that after a certain number of progressive top-k SQL operations have been executed under sel_i, operations associated with some other weightings for the same sel_i can be skipped.

Recall that the best possible ranking of fyg[R_o is k_o þ 1, since jfyg[R_oj¼ k_o þ 1. Therefore, the lower bound of Dk, denoted as Dk_L equals 1. So, this time, we project Dk_L onto slope Pen_{min} in order to determine the corresponding maximum feasible Dw value. We name that value as Dw^f. For any $Dw > Dw^f$, it means "fyg[R_o has $Dk < Dk_L$", which is impossible. As our algorithm is designed to examine weightings in their increasing order Dw values, when a weighting w_j 2 S_w has jjw_j $w_o$jj > Dw^f, topkðsel$_i$;w~$_j$; stopping-conditionÞ and all subsequent progressive top-k

ID	D	E	F
P1	60	200	70
P2	100	250	90
P3	70	280	80
P4	90	300	90
P7	80	300	100
P8	60	200	60

Fig3.Table T3

SQL operations topkðseli;w~l;stopping-conditionÞ where l > j þ 1 could be skipped.

Reuse Fig. 5 as an example. By projecting Dk_L ¼ 1 onto the slope $Pen_{Q0}1o$, we could determine the corresponding

Dw^f value. So, when the algorithm finishes executing a progressive top-k SQL operation for weighting $w_{\sim 2}$, the algorithm can skip all the remaining weightings and proceed to examine the next selection condition.

As a remark, we would like to point out that the pruning power of this technique also increases while the algorithm proceed

IV. METHODOLOGY

4. ANSWERING WHY-NOT QUESTIONS ON TOP-K SPJA QUERIES

In this section, we extend the discussion to why-not top-k SQL queries with GROUPBY and AGGREGATION.

The Problem and The Explanation Model Initially, a user issues an original top-k SPJA query Q_oðSPJA$_o$;k_o;$w_{\sim o}$Þ on a dataset D. After she gets the result R_o, she may pose a why-not question about a set of missing groups Y ¼fg1;...;gIg (l 1), where g_i has the same set of projection attributes as Q_o. Then, the system returns the user a refined query Q^0ðSPJA^0;k^0;$w_{\sim 0}$Þ, whose result R^0 includes Y and R_o, i.e., fY [R_og R^0. If there are indeed no refined queries Q^0 that can include Y and R_o, the system will report to the user about her error.

Algorithm 1. Answering a Why-not Top-k SPJ Question

```
Input:
  Original query and the
  expected tuple
1: Obtain QS^0 and j value.
2: if QS^0 does not exist then
3:      return "cannot answer the why-not
question";
4: end if
5: switch(choice)
6: case 1:
7:      spja constructs are changed.
8:      if Q_result equal to j then
9:              calculate penalty;
10:             if penalty <=min_penalty then
11:                     min_penalty =
penalty;
12:             end if
13:     end if
14:             Return query with min_penalty
15:case 2:
16:     k value alone is changed
17:     if Q_result equal to j then
18:             calculate penalty;
19:             if penalty <=max_penalty then
20:                     max_penalty =
penalty;
21:             end if
22:     end if
23:             Return query with min_penalty
24:case 3:
25:     weight values are changed
26:     if Q_result equal to j then
27:             calculate penalty;
28:             if penalty <=max_penalty then
29:                     max_penalty =
penalty;
30:             end if
31:     end if
32:             Return query with min_penalty
33:case 4:
34:     all are changed reandomly
35:     if Q_result equal to j then
36:             calculate penalty;
37:             if penalty <=max_penalty then
38:                     max_penalty =
penalty;
39:             end if
40:     end if
41: :Return query with min_penalty
```

This algorithm will give the query that includes the expected tuple with least penalty.

CONCLUSION

In this paper, we have studied the problem of answering why-not questions on top-k SQL queries. Our target is to give an explanation to a user who is wondering why her expected answers are missing in the query result. We return to the user a refined query that can include the missing expected answers back to the result. Our case studies and experimental results show that our solutions efficiently return very high quality solutions.

Acknowledgment

This work is supported by INTEL MULTICORE LAB in Department of Computer Science and Engineering, Thiagarajar College of Engineering, Madurai.

REFERENCES

[1] A. Chapman and H. Jagadish, "Why not?" in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2009, pp. 523–534.

[2] J. Huang, T. Chen, A.-H. Doan, and J. F. Naughton, "On the provenance of non-answers to queries over extracted data," in Proc. VLDB, 2008, pp. 736–747.

[3] M. Herschel and M. A. Hernandez, "Explaining missing answers to SPJUA queries," in Proc. VLDB, 2010, pp. 185–196.

[4] Q. T. Tran and C.-Y. Chan, "How to ConQueR why-not questions," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2010, pp. 15–26

[5] Z. He and E. Lo, "Answering why-not questions on top-k queries," in Proc. IEEE 28th Int. Conf. Data Eng., 2012, pp. 750–761

[6] I. Md. Saiful, Z. Rui, and L. Chengfei, "On answering why-not questions in reverse skyline queries," in Proc. IEEE 28th Int. Conf. Data Eng., 2013, pp. 973–984.

[7] Z. He and E. Lo, "Answerin.g why-not questions on top-k queries,"